MONTSE MEDINA, NIELS JOUBERT

LISZT USER MANUAL

17 MAY 2010

This tutorial is intended for programmers who are familiar with mesh based, PDE's solvers coding environment, and are interested in learning Liszt.

Contents

1 Starting a new Liszt Project 5

*1.1 Source Files: *.scala* 5

1.2 Configuration File: liszt.cfg 6

1.3 Loading a Mesh 6

2 Variables and Values 6

3 Functions in Liszt 7

4 Objects in Liszt 7

5 Liszt Value Types 8

5.1 Liszt Value Types: Vector 8

5.2 Liszt Vector Functions 9

6 Liszt Value Types: Matrix 10

7 Operators for Liszt Value Types 11

8 Liszt Mesh Types 11

8.1 Mesh Topology Functions 11

8.2 Iteration Over Mesh Type Sets 11

8.3 Boundary Sets 12

8.4 Special Mesh Type: Exterior Cell 12

9 Reference Type: Field 13

9.1 Initializing Fields 13

9.2 Field Reads 13

9.3 Field Writes 14

10 Global Variables 14

11 Printing 14

12 Traits and Mixins 15

13 Sparse Matrices 15

13.1 Indices 16

13.2 Linear Systems 16

13.3 Creating Indices 17

13.4 Mixins 18

13.5 Non-zeroes 19

14 *Example Project: Scalar Convection* 19

1 Starting a new Liszt Project

Welcome to Liszt, a high performance DSL[1] for mesh-based programming. The Liszt DSL is embedded in the Scala programming language. Scala is a Java variant and runs on the Java Virtual Machine, but takes an approach closer to that of Python and Haskell than Java and C++. Liszt tries to draw from both and follows a high level programming model that is a hybrid of functional-inspired concepts and procedural-style C code. Regardless of your programming background, we will walk you through the Liszt language and introduce all the necessary concepts to write mesh-based codes in our DSL. Let's get started!

[1] Domain Specific Language

All Liszt code - every variable, constant and function (including main) - must be inside an object. Here's a simple Liszt Hello World program:

```
@lisztcode
object Test {
    def main() {
        Print("Hello world!")
    }
}
```

Things to note:

Semicolons are optional in Scala. You can use semicolons to separate multiple statements on a single line, but you don't need to end a statement with a semicolon if it's the only statement on a line. Note that there is no final semi-colon at the end of the class definition or the `Print` expression.

The function `main` is a member of the object `Test` and takes no arguments. All objects must be prefaced by the `@lisztcode` annotation. More on objects in section 4.

To write to standard output, you use `Print(...)`.

1.1 Source Files: *.scala

A programmer in Liszt deals with source files which contain actual code, and a configuration file to modify the runtime behavior of the code. Your source code needs to abide by two rules:

1. All files in the project should be inside the same directory.
2. Every file should import two packages, as follows:

```
import Liszt.Language._
import Liszt.MetaInteger._
```

1.2 Configuration File: liszt.cfg

The configuration file consists of a list of key-value pairs. All these key-value pairs should be separated by commas, and should be enclosed in curly braces. Arguments to the main function are places in the `liszt.cfg` file, and not passed from the command line. This file must be located in the same folder as your source files. For example:

```
{
    "runtimes": ["single"],
    "main-class": "myTest",
    "mesh-file": "../myMesh.lmesh",
}
```

The keys "runtime", "main-class" and "mesh-file" refer to the runtime option, the object with the main function, and the mesh path respectively.

Runtime setup:

The Runtime is specified in liszt.cfg. Runtime options currently supported by Liszt are "`single`", "`mpi`" and "`gpu`".

1.3 Loading a Mesh

Liszt derives its performance by deeply integrating mesh topology with computation. Liszt is specifically built for computations on meshes, thus it natively supports loading a mesh as configuration parameter. The mesh path is specified in `liszt.cfg`, and accepts the following formats:

- `*.msh` – the Fluent File format
- `*.VTK` – the Visualization Toolkit mesh formats

2 Variables and Values

Liszt, like Scala, supports both constant values (immutables) and variables (mutables). The type definition syntax varies for constant values or variables as follows:

`val` – For constant values, the syntax is as following:

```
val a : T = <my code>
```

[2]

`var` – For variable values, the declaration is as following:

```
var a : T = <my code>
```

The syntax can be further simplified by dropping the type when it can be inferred by the value assigned to: `var a = true` [3]

[2] T is the type of a

[3] a is a boolean variable, since the compiler can infer the type of a by the value being assigned to it.

```
@lisztcode
object MyLisztCode {
    def MyFunction() {
        val a = 3        // equivalent in C++ : int a = 3 ;
        a = 4             // Bug! It is not allowed to reassign a val

        var b = 3.0        // equivalent in C++ : double b = 3.0 ;
        b = 4.5

        val c : Boolean = true     // redundant
        val d = true         // equivalent in C++ : bool d = true ;
    }
}
```

3 Functions in Liszt

Liszt programs are built using function calls that takes multiple values and returns a single type. All arguments are passed by value - copied into the function - and returned by value as well. Functions must be declared inside an object. You need one `main` function in your project, from where code execution starts. You specify the object with the main function in the configuration file. Function definition syntax is:

```
def foo(x1 : T1, ..., xn :Tn) : Tr = <exp>
```

`x1` is an argument variable, `Ti` is the type of argument `xi`, `Tr` is the return type. If `<exp>` returns a value, you precede it by an "=". If it returns nothing (the equivalent of `void` in C) you can omit the "=".

Main Function Example:

```
@lisztcode
object MyLisztCode {
    def main() {         // Notice that the ''='' is omitted.

        <liszt code>
    }
}
```

Function with arguments and a return value example:

```
@lisztcode
object MyObject {
    def MyFunction( a: Int, b: Int ) : Boolean = {
        val c = ( a == b )
        return c
    }
}
```

4 Objects in Liszt

Objects in Liszt are modules[4]. An object can only be declared at

[4] You can think of modules as a C++ Namespace or a Singleton Class

the top level, so you cannot nest objects. Objects must start with `@lisztcode`. All objects are public and no constructor is required.

Object Example:

```
@lisztcode
object MyLisztCode {
    val a = 0
    def main() {
        var b = false
    }
}

@lisztcode
object Test {
    def TestFunction() {
        val b = MyLisztCode.a        // note how you call a function in a different object.
    }
}
```

Things to note:

`a` is a constant at object level (outside of any function), and can therefore be accessed by other objects and their functions.

Both `b` variables are independent and exist only inside the functions where they have been declared.

5 Liszt Value Types

Liszt values types are passed and return by value through functions. Value types may be declared as **`val`** (for constant usage) or **`var`** (for variable usage). Liszt value types include `Int`, `Float`, `Double`, `String`, `Boolean`, `Vec` and `Mat`. Please see the Liszt Language Specification for precise definitions of all the value types.

5.1 Liszt Value Types: Vector

We discuss dense vectors here - a fixed-length container of a single type of values. Vectors are defines as:

`Vec[N <: IntM, T]` — A dense vector type of length `N`, where `N` is a meta-integer. `T` is any Liszt value type. Meta-integers are integer literals preceded with underscores. You need to use them to declare vectors and matrices. You may access vectors and matrices with either normal integer value sor with meta-integers.[5]

Declaration:

[5] Meta-integers are used when Liszt expects a constant value at compile time, which allows for static checking and optimizations. You should use meta-integers when possible to enable optimizations.

```
@lisztcode
object MyLisztCode {
    val myFirstVector : Vec[_3,Float] = Vec(0.f, 0.f, 0.f)
}
```

Things to note:

"myFirstVector" has type Vec of length 3, with values of type Float.

Type definition can be dropped, leading to: `val myFirstVector = Vec(0.f, 0.f, 0.f).`

Vector Entry Read:

Reading a entry in a Vec is done by accessing the desired element by specifying the desired position with a meta-integer. Position starts with the meta-integer _0. The first four elements may be accessed by calling member variables x, y, z and w respectively. A normal integer can be used to access a vector as well, but this lookup may take a slower pathway.

```
@lisztcode
object MyLisztCode {
    val v = Vec(0, 1)

    val a = v.x        // equivalent to v(_0)
    val b = v(_1)         // equivalent to v.y
    val c = v(1) // equivalent to v(_1), but possibly slower
}
```

Vector Entry Write:

A vector declared as a **val** (constant) cannot be modified. A vector declared as a **var** may be modified.

```
@lisztcode
object MyLisztCode {
    val v = Vec(0,1)
    v = Vec(2,3)         // bug! v cannot be modified

    var vv = Vec(0,0)
    vv(0) = 1
    vv = Vec(1,1)        // ok
}
```

5.2 *Liszt Vector Functions*

We provide a cross product, dot product and normalization functions that work on vectors:

def `cross[VT](a: Vec[_3,VT], b: Vec[_3,VT]) : Vec[_3,VT]` — Returns the cross product of vectors "a" and "b" of length 3

def `dot[N <: IntM, VT](a : Vec[N,VT], b: Vec[N,VT]) : VT` — Returns the dot product of vectors "a" and "b" of length N

`def normalize[N <: IntM, VT](a : Vec[N,VT]) : Vec[N,VT]` — Returns a new vector equivalent to vector "a" normalized

6 *Liszt Value Types: Matrix*

Matrices are fixed-size 2D storage in Liszt. It is defined as:

`Mat[R <: IntM, C <: IntM, T]` — A dense matrix type of dimension `R` rows by `C` columns, where `R` and `C` are meta-integers. `T` is any Liszt value type.

Declaration:

```
@lisztcode
object MyLisztCode {
    val myFirstMatrix = Mat(Vec(0.f, 0.f, 0.f), Vec(0.f, 0.f, 0.f))
}
```

Things to note:

"myFirstMatrix" is a 2x3 dense matrix.

Each row in the matrix is declared as a Liszt Vector of length C.

Matrix Read

Access to the desired element is possible by specifying the desired 2D position with the corresponding meta-integers. The first element corresponds to the row and the second element specifies the column:

```
@lisztcode
object MyLisztCode {
    val m = Mat(Vec(1,0), Vec(0, 1))
    val a01 = m(_0,_1)
}
```

Matrix Write

A matrix declared as a **val** (constant) cannot be modified. A matrix declared as a **var** may be modified:

```
@lisztcode
object MyLisztCode {
    val m = Mat( Vec(1,0), Vec(0,1) )
    m = Mat( Vec(2,3), Vec(0.1) )     // bug! v cannot be modified

    var mm = Mat( Vec(0,0), Vec(0,0) )
    mm(_0,_0) = 1
    mm = Mat( Vec(1,0), Vec(0,1) )
}
```

7 Operators for Liszt Value Types

Please see the Liszt Language Specification for all the operators defined on Liszt's value types.

8 Liszt Mesh Types

Once Liszt imports your mesh, it makes the mesh topology available through a set of mesh types and functions. Mesh types are always declared as **`val`**, and are therefore constant. This reflects the current state of Liszt - your mesh topology is static over the course of your program. Mesh types are passed and return by value through functions. Liszt provides the following mesh types:

`Vertex` — A 0-dimension element, representing a single point on the mesh.

`Edge` — A 1-dimensional element, connecting two vertices.

`Face` — A 2-dimensional element (triangle, quadrilateral, etc.), composed of several edges.

`Cell` — A 3-dimensional element (tetrahedron, hexahedron, etc.), composed of several faces.

`Mesh` — An entire mesh. Currently the only build-in expression with type `Mesh` is `mesh`.

`Set[T <: MeshObj]` — A set of vertices, edges, faces, or cells.

8.1 Mesh Topology Functions

Liszt provides built-in functions to access mesh topology, such as `vertices(mesh)`, `edges(mesh)`, `cells(mesh)`, `faces(mesh)` and, for example, `faces(cell)`. Please see the Liszt Language Specification's section on Mesh Topology Functions for a full list.

8.2 Iteration Over Mesh Type Sets

Iteration over set of elements does not guarantee a sequential order. However, each element in the set is guaranteed to execute the code specified inside the loop. For example:

```
@lisztcode
object MyLisztCode {
    for ( c <- cells(mesh) ) {
        for ( f <- faces(c) ) {
            <my Code 1>
        }
        for ( cc <- cells(c) ) {
```

```
            <my Code 2>
        }
    }
}
```

Things to note:

"c" and "cc" are of type Cell

"f" is of type Face

<my Code 1> will be executed for all faces around every cells in the mesh

8.3 Boundary Sets

Boundary sets allow you to access subsets of the total mesh topology, as defined by the sets in your input mesh file. Boundary sets are declared as follows:

`BoundarySet[A <: MeshObj]` — `A` is one of `Vertex`, `Edge`, `Face` or `Cell`

Boundary sets are special sets that contain boundary elements. They must always be declared as **`val`**. The elements contained in the boundary set are specified in the mesh file, where each boundary has a characteristic string name chosen by the user. A usage example follows here. In this example, "default-interior" must be a string specified in the mesh file.

```
@lisztcode
object MyLisztCode {
    val boundary = BoundarySet[Face]("default-interior")
}
```

8.4 Special Mesh Type: Exterior Cell

The exterior cell is a special cell element with cell global id equal to zero. This allows you to find the outside of the mesh. For example:

```
@lisztcode
object MyLisztCode {
    val boundary = BoundarySet[Face]("wall")

    for ( f <- wall ) {
        val c0 = inside(f)
        val c1 = outside(f)

        if ( ID(c0) == 0 ) {
            // c0 is the exterior cell and c1 is interior
        } else if ( ID(c1) == 0 ) {
            // c1 is the exterior cell and c0 is interior
        } else {
            // c0 and c1 are interior cells
```

```
        }
    }
}
```

9 Reference Type: Field

Fields are data structures that will store data of type T for each element of mesh type A in the mesh. Fields support synchronization and are, therefore, updated during set iteration process. Fields must be declared at object level scope. Fields are passed to functions as reference and cannot be returned from functions. A field has to be declared as **val**. However, the stored value T behaves as **var** and therefore can be modified. They are defined as:

`Field[A <: MeshObj, T]` — A is one of Vertex, Edge, Face or Cell. T is any Liszt value type.

A usage example of fields are:

```
@lisztcode
object MyLisztCode {
    val cellField = FieldWithConst[Cell,Int](0)
    for ( c <- cells(mesh) ) {
        cellField(c) = ID(c)
    }
}
```

9.1 Initializing Fields

There are two way of initializing fields:

`FieldWithConst` – Initializes the field with a constant passed as an argument

`FieldWithLabel` – Every element in the field is initialized with a different value specified in the label input argument "position"

```
@lisztcode
object MyLisztCode {
    val cellField = FieldWithConst[Cell,Float](0.f)
    val positionField = FieldWithLabel[Vertex,Vec[_3,Float]]("position")
}
```

9.2 Field Reads

You read a value stored for a mesh element in a Field by specifying the associated topology element for which the field value must be read. The return type is T, the type stored in the field. For example:

```
@lisztcode
object MyLisztCode {
    val positionField = FieldWithLabel[Vertex,Vec[_3,Float]]("position")
    for ( v <- vertices(mesh) ) {
        val pos = positionField(v)
    }
}
```

9.3 Field Writes

Fields cannot be written to and read from in the same loop.[6]

[6] This limitation enables much greater parallelization to be automated.

```
@lisztcode
object MyLisztCode {
    val cellField = FieldWithConst[Cell,Int](1)
    for ( c <- cells(mesh) ) {
        cellField(c) += ID(c)
    }
}
```

10 Global Variables

Global Variables are declared no differently than local variables in Liszt. Any **var** defined outside a **for** loop and updated inside the loop will become a global variable. Global variables can only be either read or written in the same loop. For example:

```
@lisztcode
object MyGlobalVar {
    var numCells = 0
    def main() {
        for ( c <- cells(mesh) ) {
            numCells += 1
        }
    }
}
```

11 Printing

Print function can take any number of arguments separated by commas, and outputs it to the standard output - usually your command line.

`Print(as : Any*) : Unit` — Output all arguments, newline terminated.

For example:

```
@lisztcode
object MyPrintTest {
    var numCells = 0
    def main() {
        for ( c <- cells(mesh) ) {
            numCells += 1
```

```
        }
    }
    Print("number of cells in mesh: ", numCells)
}
```

12 Traits and Mixins

Liszt supports a subset of the trait and mixin interface that scala provides, which lets you reuse common pieces of code by mixing them into your objects. To create a reusable piece of code you can declare a **trait**:

```
@lisztcode trait A {
    var a = 1
}
@lisztcode trait B {
    val b = 2
}

@lisztcode object C extends A with B {
    def main() {
        D.a = 3
        Print(a," ",b," ",D.a) //prints "1 2 3"
    }
}
@lisztcode object D extends A
```

Here we declare two traits `A` and `B` and then mix them into a single object `C`. You can mix the same trait into multiple objects, allowing you to reuse the same code in multiple places, as we have done with `D`. Liszt also allows traits to be parameterized:

```
@lisztcode trait Mult[N <: IntM] {
    def run(a : Float, b : Vec[N,Float]) : Vec[N,Float] = a * b
}

@lisztcode object Mult3 extends Mult[_3]
@lisztcode object Main {
    def main() {
        Print(Mult3.run(Vec(1.f,2.f,3.f))) //prints "[2,4,6]"
    }
}
```

Here the parameter list `[N <: IntM]` indicates that N must be a meta-integer (`IntM`). `Mult3` then supplices `Mult` with the meta-integer `_3` to create an instance of the `Mult` trait for use with vectors of size 3. Parameterized traits allow you to write more general code and use it in multiple places.

13 Sparse Matrices

Liszt supports solving linear-systems `Ax = b` defined over the liszt mesh by interfacing with external solvers. This section describes how

linear systems are described and solved in liszt.

13.1 Indices

To get started, we introduce an abstract data type called Index. An Index refers to a single row or column of a sparse matrix, or it refers to a single entry in a sparse vector. An index is an abstract data type; it not possible to perform integer math on an index. We will introduce a way to construct indicies shortly, but for now let's assume you can obtain indicies and focus on how to use them. Here are some example uses: (assume `i1,i2,i3 ...` are indicies, `v1,v2,v3 ...` are sparse vectors, and `m1,m2,m3 ...` are sparse matrices. Variables are explicitly typed for clarity, but these declarations are optional in Scala).

You can access vectors like so:

```
val f0 : Float = v1(i1)
val f1 : Float = m1(i1,i2)
val index_vector : Vec[_3,Index] = Vec(i3,i4,i5)

//gather a small dense vector from a sparse vector
val f2 : Float3 = v1(index_vector)

val index_vector2 : Vec[_3,Index] = Vec(i6,i7,i8)

//gather a small dense matrix that is the product of a vector of row indicies
//and a vector of column indicies
val f3 : Float3x3 = m1(index_vector,index_vector2)
```

Writing to these objects looks very similar:

```
v1(i1) = 1.f
m1(i1,i2) = 2.f
v1(index_vector) = Vec(1.f,2.f,3.f)
m1(index_vector,index_vector2) = f3
```

13.2 Linear Systems

We combine these new types into a linear system, which represents the equation: `Ax = b`. You create a linear system object by mixing in the LinearSystem trait when you want to use sparse matrix solvers.

```
trait LinearSystem {
    type X <: Vector
    type B <: Vector
    type A <: Matrix

    //constructors for the elements of the linear system
    //calling each function returns a new object that can be used in a call to solve
    def x() : X
    def b() : B
    val A() : A {
        type RowIndex = B.Index
        type ColIndex = X.Index
    }
```

```
    def solve(A : A, x : X, b : B) //solve for x given A and b
    def nonzeros( non_zero_constructor : (A,X,B) => Any) //method to create non-zeros, see below for detail
}
```

Notice that the row and column indicies are different types, and that `RowIndex = B.Index`, and `ColIndex = X.Index`. This ensures that all accesses to these objects have the right index types. The solve method actually invokes a solver, modifying `x` based on the values in `A` and `b`.

13.3 Creating Indices

Let us now revisit the way to construct Index values by first recalling how we might create indices by hand in a non-dsl code. If we had a simple first-order FEM code, our rows and columns of ours sparse matrix map one-to-one with vertices in the mesh. so me might write something like so:

```
type Index = Int
def rowIndex(v : Vertex) : Index = ID(v)
def colIndex(v : Vertex) : Index = ID(v)
```

A more complicated code might use linear transformations of the ids of mesh elements to achieve the same effect, mapping mesh topology to integer indicies.

In Liszt, we retain the concept of mappings from mesh topology to indicies, but we make the mappings abstract. In liszt, you write:

```
@lisztcode
object MyLinearSystem1 extends LinearSystem {
    def rowIndex(v : Vertex) : A.RowIndex = AutoIndex
    def colIndex(v : Vertex) : A.ColIndex = AutoIndex
}
```

Instead of implementing these methods using linear transormation of IDs, Liszt will automatically implement these methods. Liszt will assume that for each method, and for each set of unique arguments to the method, a unique Index object should be created. Let's consider a more complicated 2nd order FEM case where indicies exists for edges and vertices. Here we just add additional methods to return indicies for the edges:

```
@lisztcode
object MyLinearSystem2 extends LinearSystem {
    def rowIndex(v : Vertex) : A.RowIndex = AutoIndex
    def rowIndex(e : Edge) : A.RowIndex = AutoIndex

    def colIndex(v : Vertex) : A.ColIndex = AutoIndex
    def colIndex(e : Edge) : A.ColIndex = AutoIndex
}
```

You can also return `Vec[N,Index]` objects, which are useful if you are solving a vector field:

```
@lisztcode
object MyLinearSystem1Vec extends LinearSystem {
```

```
    def rowIndex(v : Vertex) : Vec[_3,A.RowIndex] = AutoIndex
    def colIndex(v : Vertex) : Vec[_3,A.ColIndex] = AutoIndex
}
```

The arguments to these methods must be mesh topology, and the return types must be indicies.

13.4 Mixins

We realize that declaring these mappings from topology to index is tedious and verbose, especially in the simple FEM cases. To work around this, we provide mixin traits for the most common cases that already are set up correctly.[7] You can think of these mixins as already pre-built recipies that you can pick and choose when you need. Here is an example for first-order FEM on triangles:

[7] This library is still in development and is not yet complete

```
trait Trangle1 extends LinearSystem {
    def rowIndex(v : Vertex) : A.RowIndex = AutoIndex
    def colIndex(v : Vertex) : A.ColIndex = AutoIndex

    def rowIndices(f : Face) = Vec( rowIndex(vertex(f,0)),
                                    rowIndex(vertex(f,1)),
                                    rowIndex(vertex(f,2)))
    def colIndices(f : Face) = Vec( colIndex(vertex(f,0)),
                                    colIndex(vertex(f,1)),
                                    colIndex(vertex(f,2)))

    def triangle1Nonzeros() //allocates non-zeros for first-order triangular FEM
}
```

Notice that we also provide two helper functions rowIndicies, and colIndicies that produce vectors of indicies for the canonical triangluar element. Given a triangle "f" and a linear system "l" you might use them like so:

```
//declare linear system
@lisztcode
object LS extends Triangle1
object Main {
    val A = LS.A()
    for(f <- faces(mesh) {
        //gather dense matrix from sparse matrix
        val matrix : Float3x3 = A(LS.rowIndicies(f),LS.colIndicies(f))
        //update dense matrix locally
        performUpdate(matrix)
        //scatter dense matrix back into sparse matrix
        A(LS.rowIndicies(f),LS.colIndicies(f)) += matrix
    }
}
```

We provide traits for the most common FEM types, so for the most part simple codes will only need to call a few functions to access the matrix.

13.5 Non-zeroes

Finally, we need a way to declare where non-zero entries are in these objects. Declaring non-zero entries explicitly ensures that the solvers are always working with the minimum number of non-zero entries. Otherwise we would need to rely on compiler analysis which may be overly conservative and lead to slower solvers.

For this we provide the nonzeroes() method in LinearSystem. This is called when you want to change the layout of non-zeroes in the LinearSystem. For 1st order FEM , you might call it like so:

```
l.nonzeroes {
(A,x,b) =>
    for(v <- vertices) {
        x(l.colIndex(v))
        b(l.rowIndex(v))
    }
    for(f <- mesh) {
        for(v1 <- vertices(f)) {
            for(v2 <- vertices(f)) {
                A(l.rowIndex(v1),l.rowIndex(v2))
            }
        }
    }
}
```

Any entry referenced in the block given to nonzeroes will be presumed to be non-zero. LinearSystems are allocated assuming that all values are zero. You must first call nonzeroes before using the linear system. When nonzeroes is called, all old non-zero values in any object derived from the linear system are invalidated. This allows you to change the format of the matrix later on in the program (if, for instance, you want to change the order of some elements). For the common FEM cases, an already implemented nonzeroes helper method will be provided in the mixin trait, so you will only need to call it once to initialize the matrix.

14 Example Project: Scalar Convection

See the bundled Scalar Convection code at: `examples/scalar_convection`